EE 590 A
Applied High-Performance GPU Computing

LECTURE 5 : Thursday, October 25, 2018

Instructor: Dr. Colin Reinhardt

University of Washington - Professional Masters Program

Autumn 2018

Lecture 5 : Outline

- HW3 Discussion and Examples
- Intel VTune Amplifier 2018 Tour
  - with In-class Exercise 5a
- SIMD/SPMD/SIMT, OpenCL, and Intel
- MMUL OPT5: blocked/tiled/partitioned
- In-class Exercise 5b

Reminders

- Office hours this Sunday 3-5 PM
- HW-4: Due Thursday 11/8 by 5:59 PM
  - will be posted by Fri 11/26 midnight

HW3 Discussion and Examples

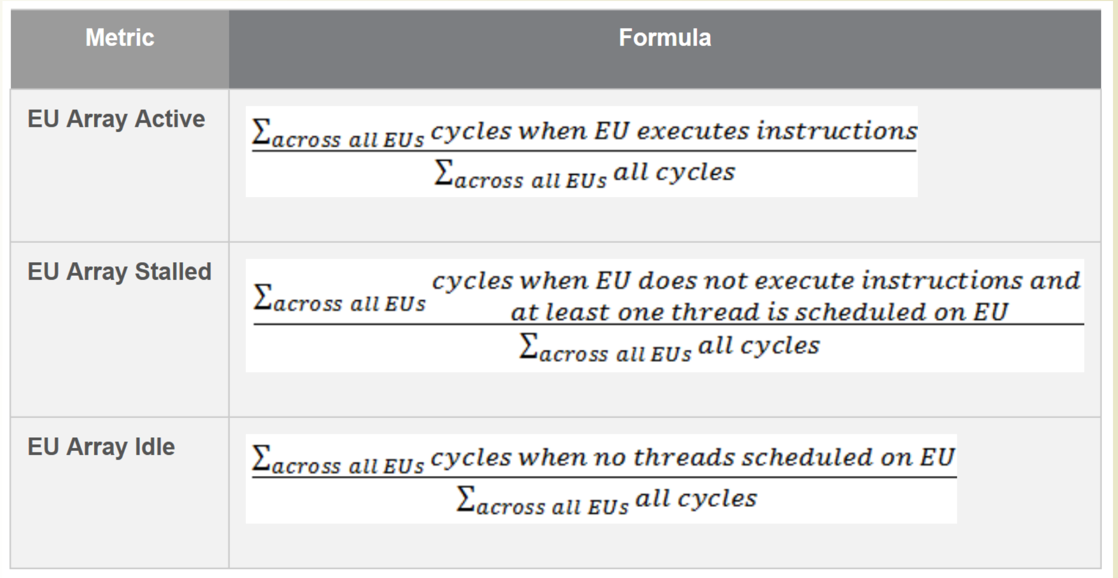
- Using OpenCL C printf() in kernels
- User-defined types (UDTs)
  - Alignment of built-in Types (C Spec, 6.1.5)
    - Unions and Structs (C Spec, 6.11.1)
      - `__attribute__((aligned (x)))`
      - `__attribute__((packed))`

In-class Exercise 5a

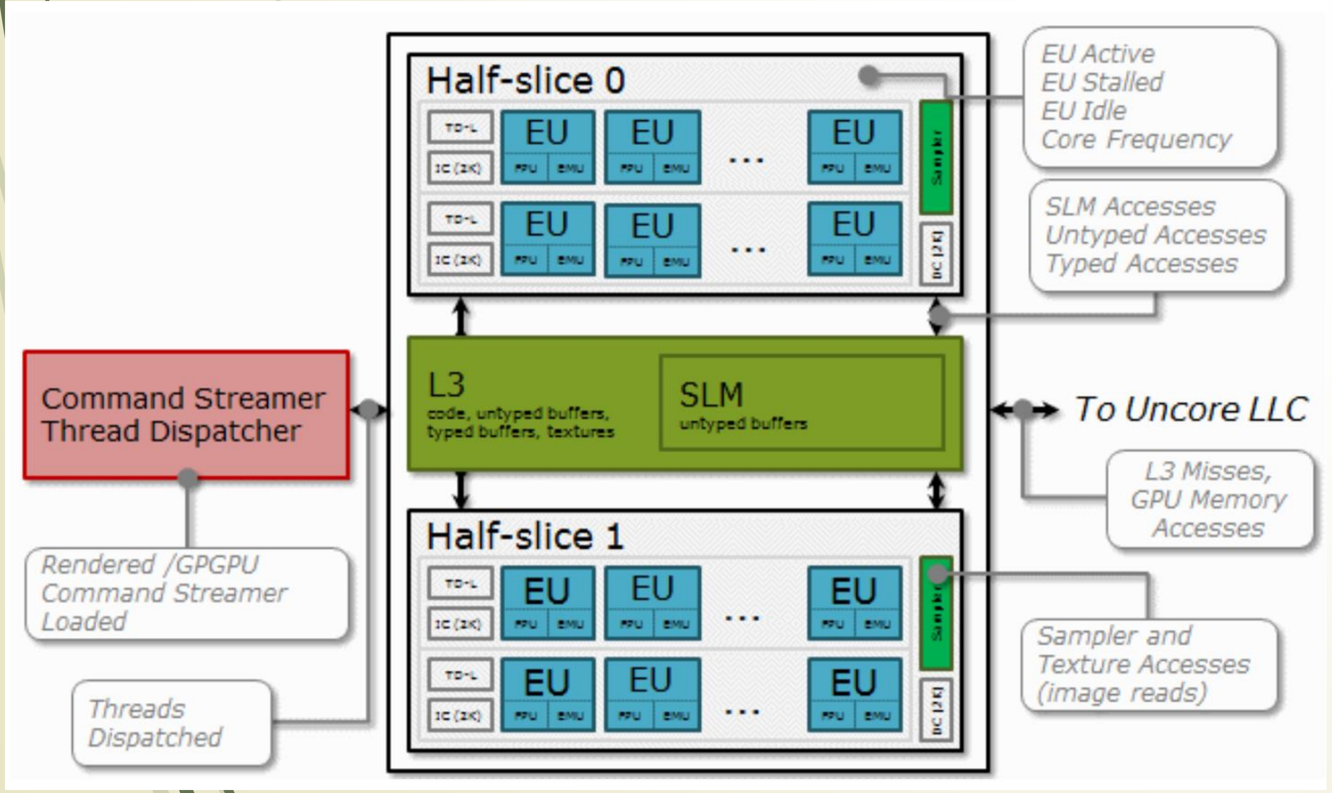
Introduction to Intel VTune Amplifier 2018

VTune GPU Metrics

- Requires Administrator privileges



Metric	Formula
EU Array Active	$\frac{\sum_{across\ all\ EUs} cycles\ when\ EU\ executes\ instructions}{\sum_{across\ all\ EUs} all\ cycles}$
EU Array Stalled	$\frac{\sum_{across\ all\ EUs} cycles\ when\ EU\ does\ not\ execute\ instructions\ and\ at\ least\ one\ thread\ is\ scheduled\ on\ EU}{\sum_{across\ all\ EUs} all\ cycles}$
EU Array Idle	$\frac{\sum_{across\ all\ EUs} cycles\ when\ no\ threads\ scheduled\ on\ EU}{\sum_{across\ all\ EUs} all\ cycles}$



VTune Amplifier provides platform-specific presets of the hardware metrics. All presets collect data about execution units (EUs) activity: EU Array Active, EU Array Stalled, EU Array Idle, Computing Threads Started, and Core Frequency.

- **Overview** event set also includes metrics that track general GPU memory accesses such as Memory Read/Write Bandwidth, GPU L3 Misses, Sampler Busy, Sampler Is Bottleneck, and GPU Memory Texture Read Bandwidth. These metrics can be useful for both graphics and compute-intensive applications.
- **Compute Basic (with global/local memory accesses)** event group also includes metrics that distinguish accessing different types of data on a GPU: Untyped Memory Read/Write Bandwidth, Typed Memory Read/Write Transactions, SLM Read/Write Bandwidth, Render/GPGPU Command Streamer Loaded, and GPU EU Array Usage. These metrics are useful for compute-intensive workloads on the GPU.
- **Compute Extended** event group includes metrics targeted only for GPU analysis on the Intel processor code name Broadwell and higher. For other systems, this preset is not available.
- **Full Compute** event group is a combination of the **Overview** and **Compute Basic** event sets.

SIMD, SPMD, SIMT

Start with definitions of the acronyms:

- SIMD: Single-Instruction Multiple-Data
- SPMD: Single-Program Multiple-Data
- SIMT: Single-Instructions Multiple Threads

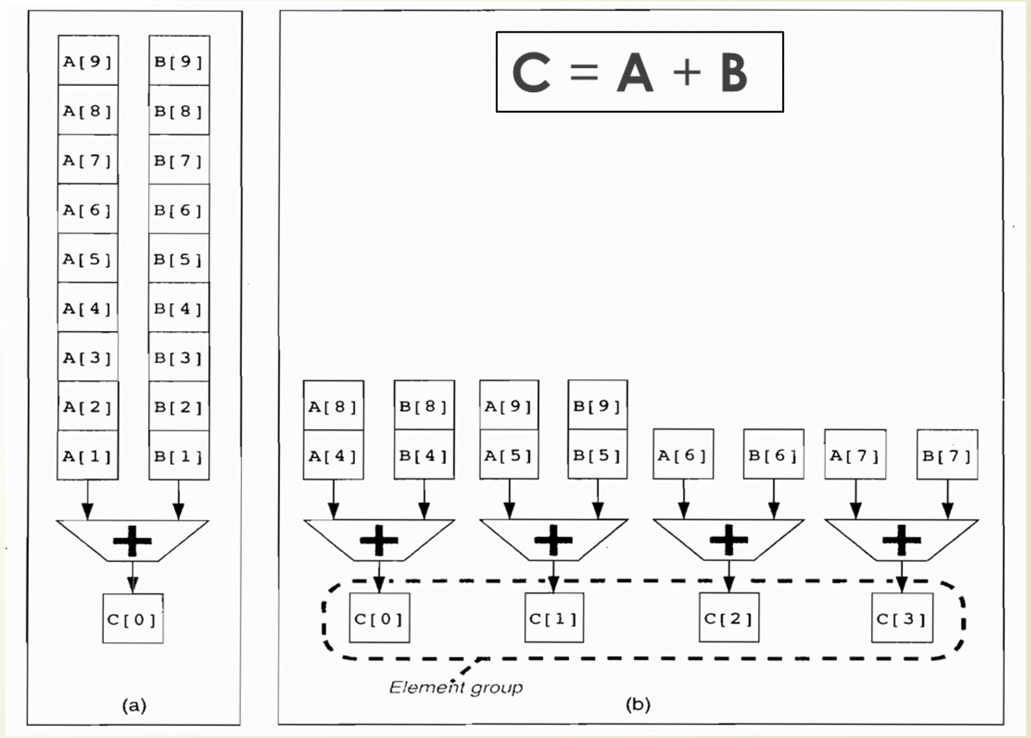
To understand these concepts requires we take a look at the hardware…

First there was SIMD

- Flynn's Taxonomy (1966)
- Originally basis for vector supercomputers in early 1970-1980s (Cray)
- older SIMD architectures: "long vector" 64- 64K vector lengths
- newer SIMD architectures: "short vector" 2-16 Words

- **Data-level parallelism** model

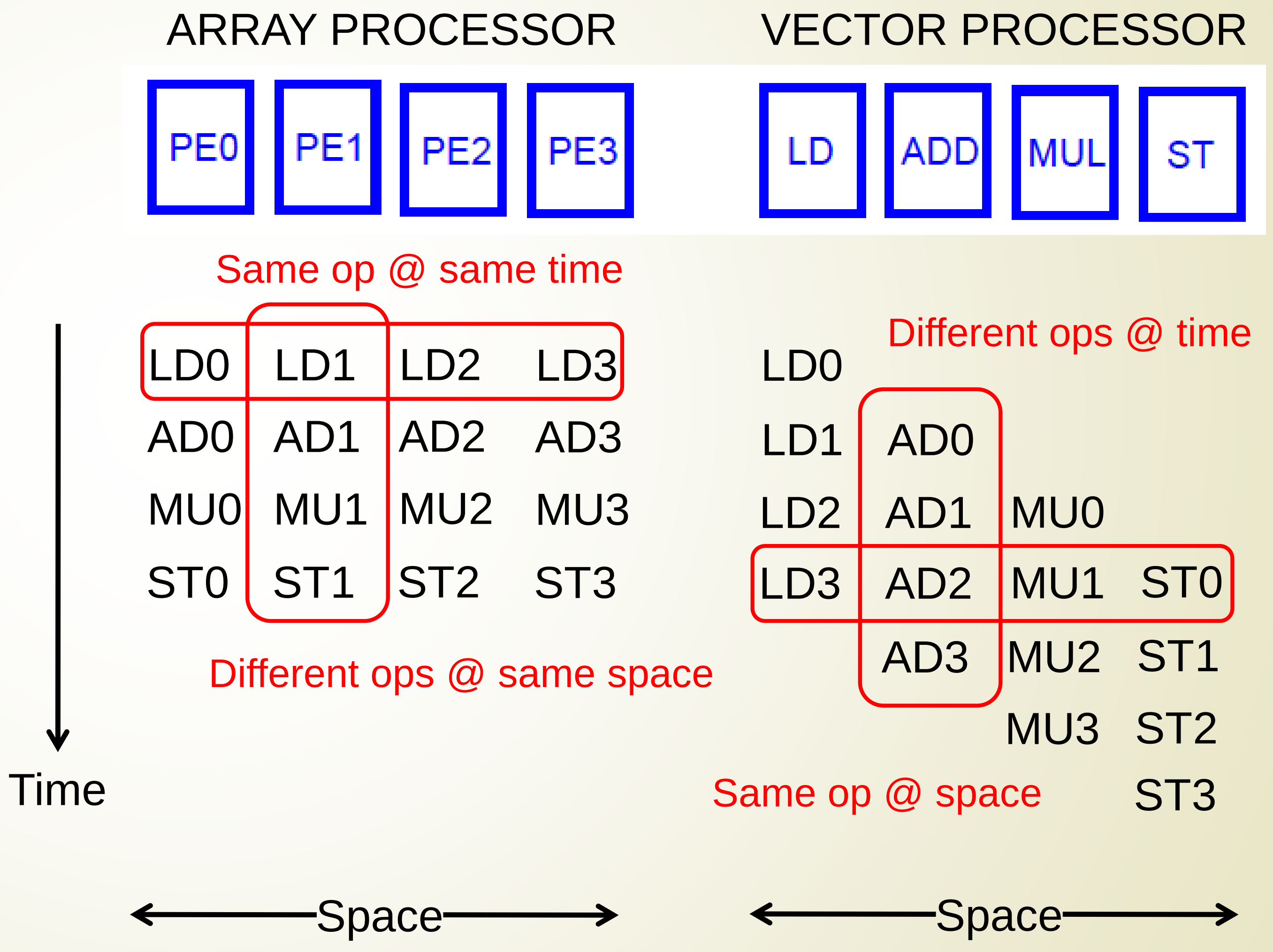
- **vector** versus **array** processors
  - pipelining limited functional units, or
  - multiple general functional units



Array vs. Vector Processors

Instruction Stream

```
LD    VR ← A[3:0]
ADD   VR ← VR, 1
MUL   VR ← VR, 2
ST    A[3:0] ← VR
```



PE0	PE1	PE2	PE3
LD0	LD1	LD2	LD3
AD0	AD1	AD2	AD3
MU0	MU1	MU2	MU3
ST0	ST1	ST2	ST3

LD	ADD	MUL	ST
LD0			
LD1	AD0		
LD2	AD1	MU0	
LD3	AD2	MU1	ST0
	AD3	MU2	ST1
		MU3	ST2
			ST3

Definitions:
VR: vector register
LD: LOAD instruction (from mem)
ADD: ADD instruction
MUL: Multiply instruction
ST: STORE instruction (to mem)

GPUs are SIMD Engines Underneath

- The instruction pipeline operates like a SIMD pipeline (e.g., an array processor)
- However, the programming is done using **threads**, NOT SIMD instructions
- We will consider a simple parallelizable loop code example

```
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];
```

- Need to distinguish between
  - Programming Model (Software)

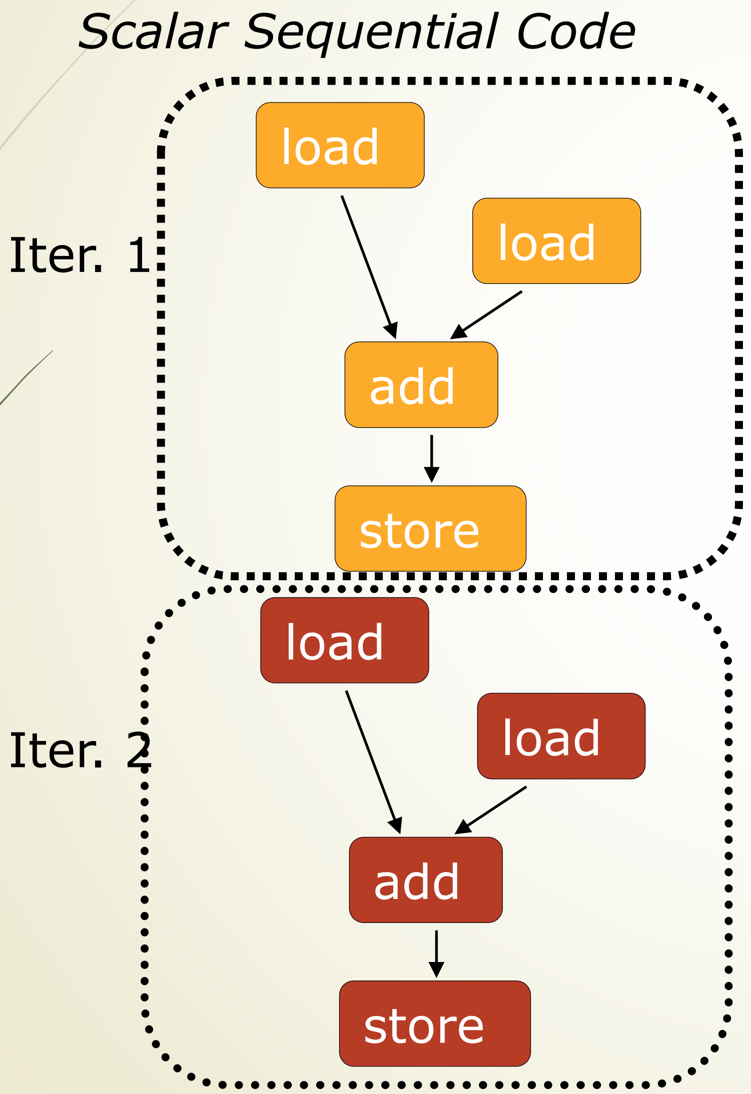
    vs.
  - Execution Model (Hardware)

Programming Model vs. Hardware Execution Model

- Programming Model refers to how the programmer expresses the code
  - E.g., Sequential (von Neumann), Data Parallel (SIMD), Multi-threaded (MIMD), Single-program multiple-data (SPMD), …
- Execution Model refers to how the hardware executes the code underneath
  - E.g., Dynamic Out-of-order execution, Vector processor, Array processor, Multiprocessor, Multithreaded processor, …
- Execution Model can be very different from the Programming Model
  - E.g., von Neumann model implemented by an OoO processor
  - E.g., SPMD model implemented by a SIMD processor (**GPUs**)

How Can You Exploit Parallelism Here?

```
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];
```



Consider three programming options to exploit the instruction-level parallelism present in this sequential code:

1. Sequential (SISD)

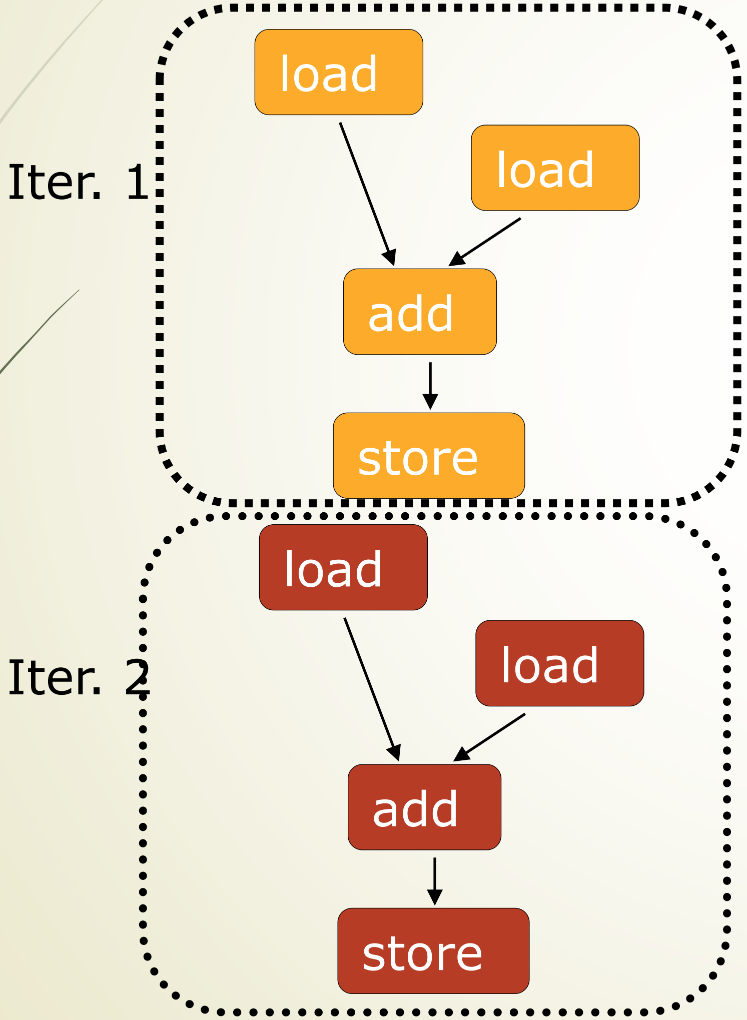
2. Data-Parallel (SIMD)

3. Multithreaded (MIMD/SPMD)

Prog. Model 1: Sequential (SISD)

```
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];
```

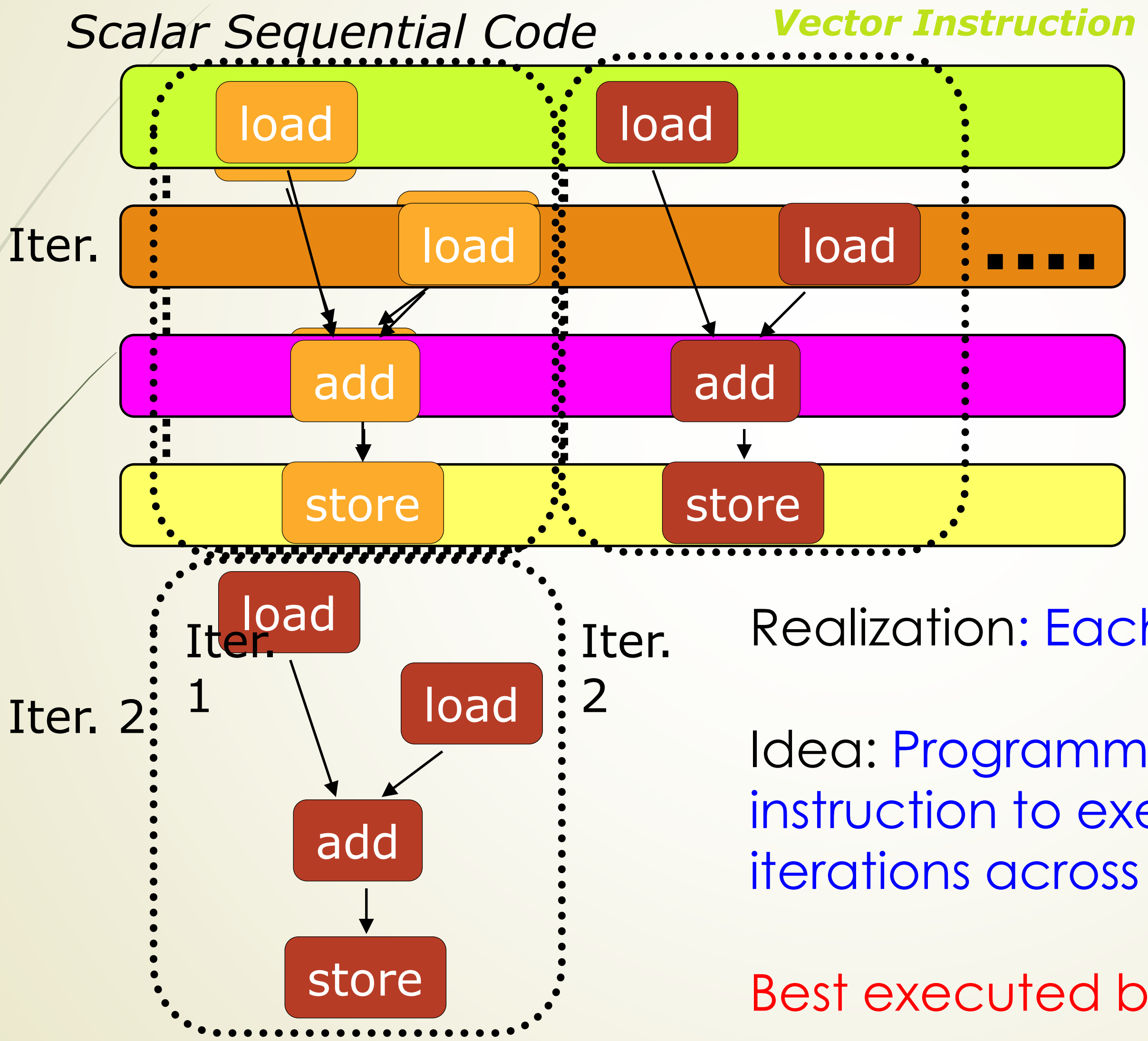
Scalar Sequential Code



- Can be executed on
- Pipelined processor
- Out-of-order execution processor
  - Independent instructions executed when ready
  - Different iterations are present in the instruction window and can execute in parallel in multiple functional units
  - In other words, the loop is dynamically unrolled by the hardware
- Superscalar or VLIW processor
  - Can fetch and execute multiple instructions per cycle

```
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];
```

Prog. Model 2: Data Parallel (SIMD)



Vectorized Code

VLD	A → V1
VLD	B → V2
VADD	V1 + V2 → V3
VST	V3 → C

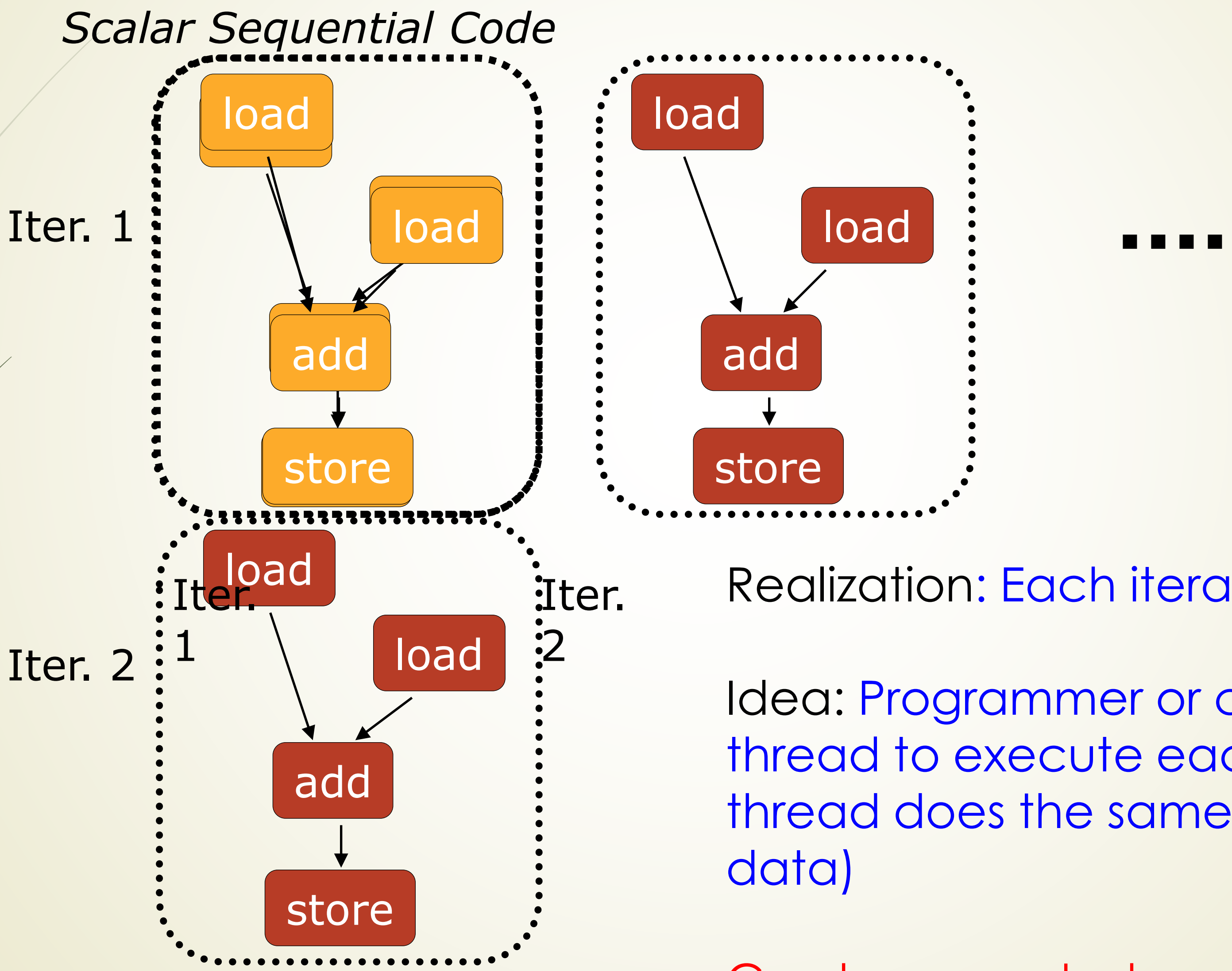
Realization: Each iteration is independent

Idea: Programmer or compiler generates a SIMD instruction to execute the same instruction from all iterations across different data

Best executed by a SIMD processor (vector, array)

```
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];
```

Prog. Model 3: Multithreaded



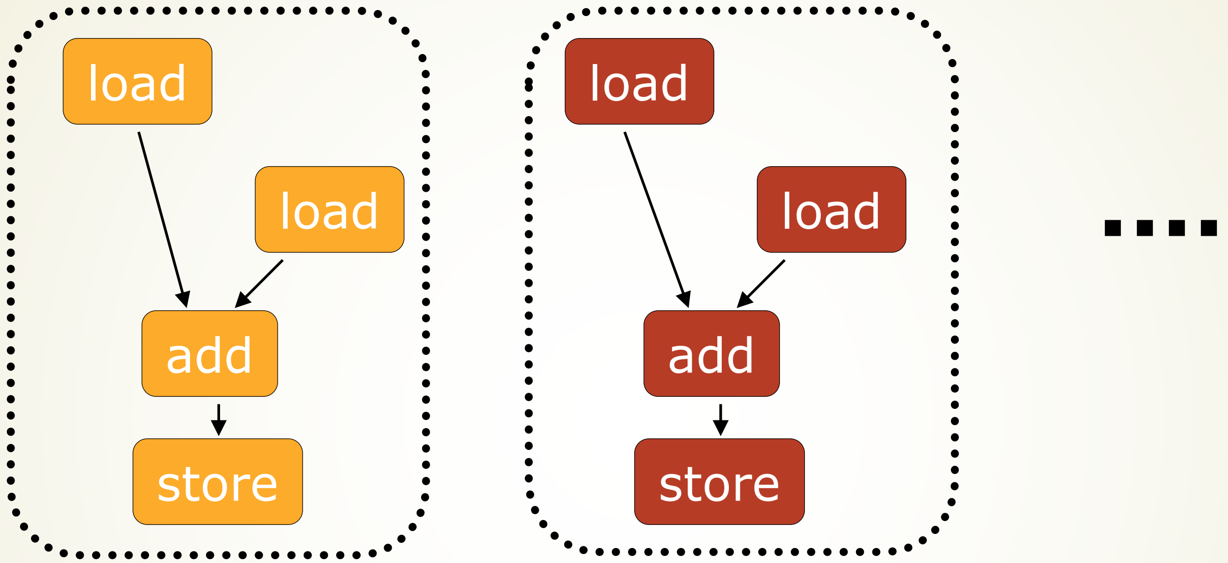
Realization: Each iteration is independent

Idea: Programmer or compiler generates a thread to execute each iteration. Each thread does the same thing (but on different data)

Can be executed on a MIMD machine

Prog. Model 3: Multithreaded

```
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];
```



Iter. 1

Iter. 2

Realization: Each iteration is independent

This particular model is also called:

SPMD: Single Program Multiple Data

Can be executed on a SIMT machine
Single Instruction Multiple Thread

A GPU is a SIMD (SIMT) Machine

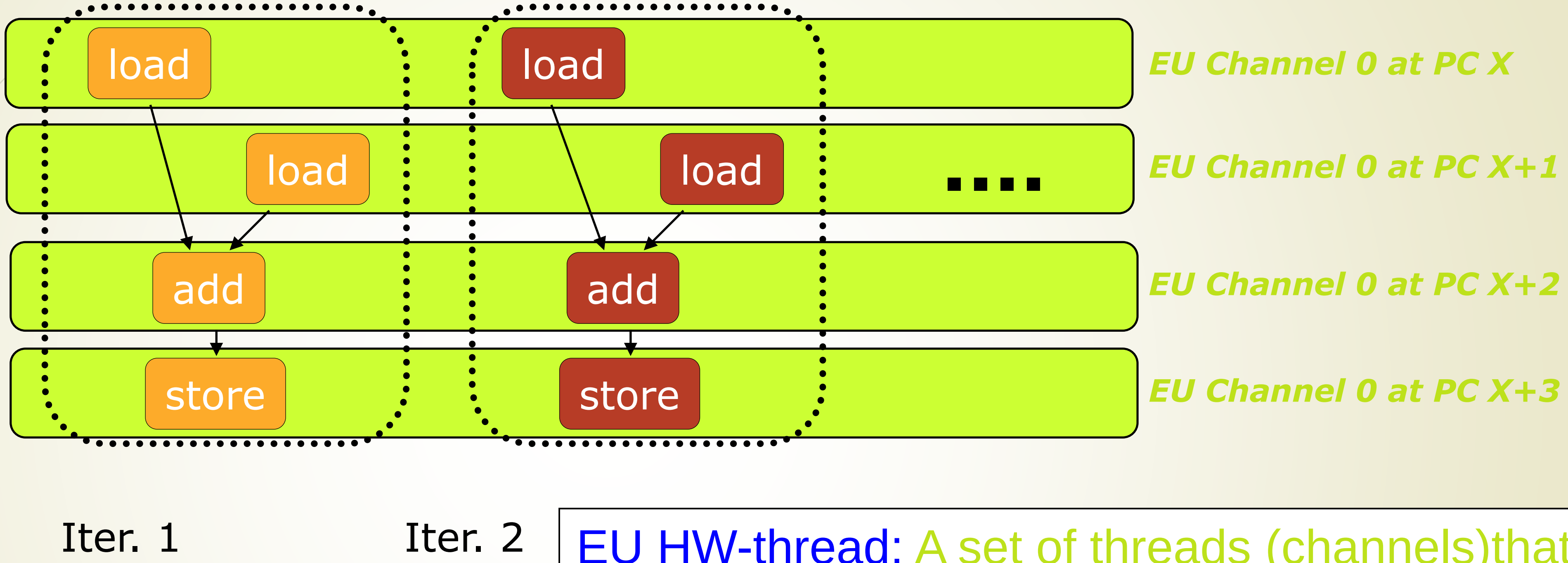
- Except it is not programmed using SIMD instructions

- It is programmed using threads (**SPMD** programming model)
  - Each thread executes the same code but operates a different piece of data
  - Each thread has its own context (state registers) (i.e., can be treated/restarted/executed independently)

- A set of threads executing the same instruction are dynamically grouped into an **EU HW-thread** (A.K.A. warp (NVIDIA), wavefront (AMD) by the hardware)
  - essentially a SIMD operation formed by hardware!

SPMD on SIMT Machine

```
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];
```



EU HW-thread: A set of threads (channels)that execute the same instruction (i.e., same PC)

This particular model is also called:

SPMD: Single Program Multiple Data

A GPU executes it using the SIMT model:
Single Instruction Multiple Thread

Intel SIMD

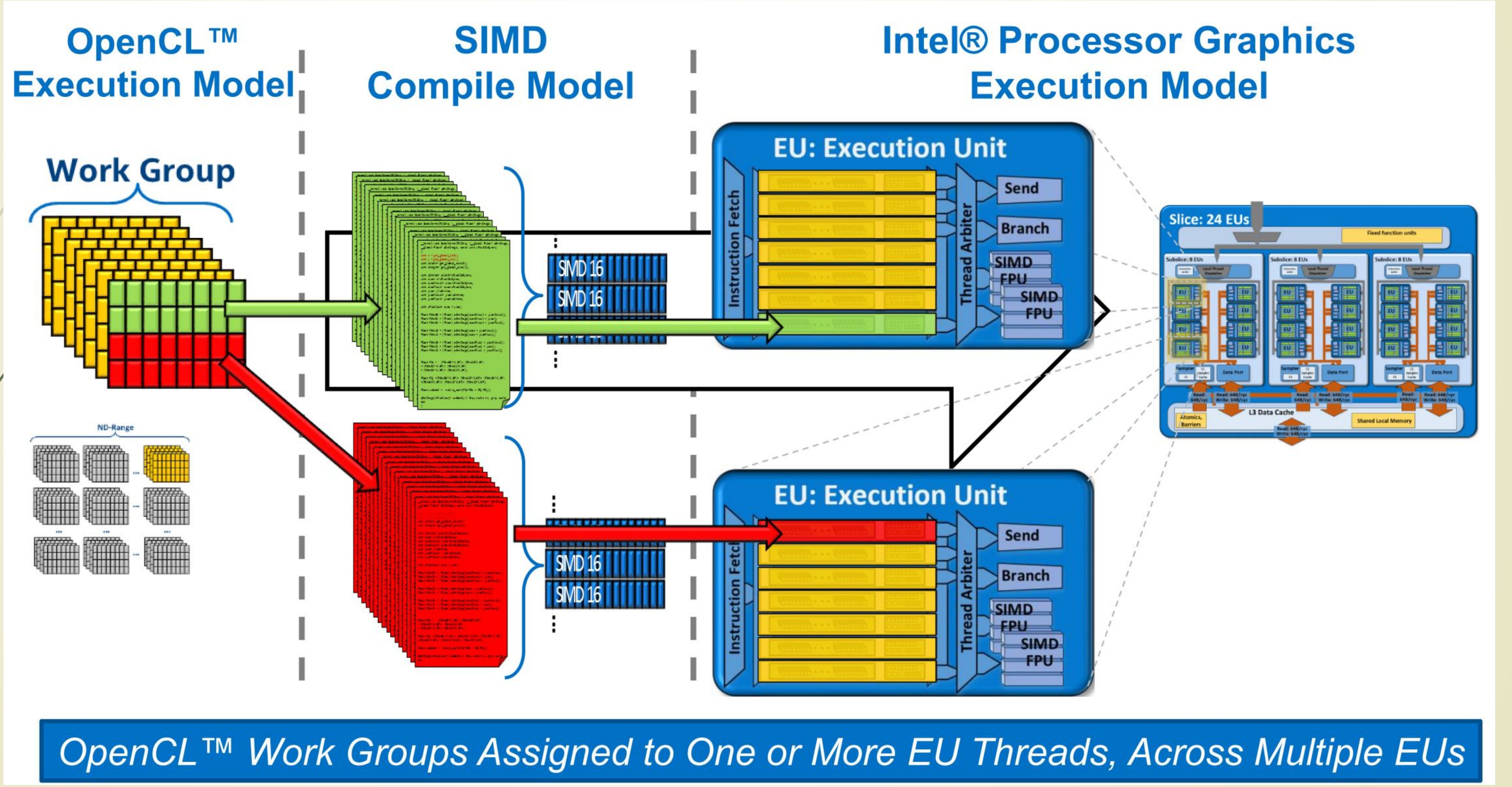
- CPU ISAs (x86, x64): SIMD-extensions: MMX, SSE, AVX, …
  - SIMD intrinsics (AVX-512): `__m512i _mm512_add_epi32(__m512i a, __m512i b)`

- Processor Graphics (GENx GPUs)
  - 2 x SIMD FPUs in each EU
  - support both floating-point and integer computation
  - parallel execute 4 x 32-bit (Word) floating-point/integer operations
    - or 8 x 16-bit (Half/Short) floating-point/integer operations
  - each SIMD FPU can complete simultaneous ADD and MULT (MAD) operations each cycle
    - Thus each EU capable of: (ADD + MUL) x 2 FPUs x SIMD-4 = 16 32-bit FLOP/cycle
  - EU General-purpose Register Files (GRFs)
    - 128 GRFs per EU HW thread
    - Each register stores 32 Bytes, accessible as SIMD 8-element vector of 32-bit (Word) data
    - 128 x 32 = 4096 bytes = 4 KB / EU HW thread

Intel: Logical versus Physical SIMD

- EU ISA and GRF are designed to support a flexible SIMD width
  - **Physical = HW** (Execution model)
    - For 32-bit data types, Gen9 FPUs are *physically* SIMD 4-wide, GRFs 8-wide.
  - **Logical = SW** (Programming-Compilation model)
    - The FPUs can be targeted with SIMD instructions and registers that are *logically* 1-wide, 2-wide, 4-wide, 8-wide, 16-wide, or 32-wide.
- GEN Compiler: scalarization and auto-vectorization passes
- If the kernel is compiled SIMD-8, then a subgroup is made up of 8 work items
  - share 4 KB of register space of an EU HW thread and execute together

Intel OpenCL SIMD Model



Intel OpenCL Extension: subgroups

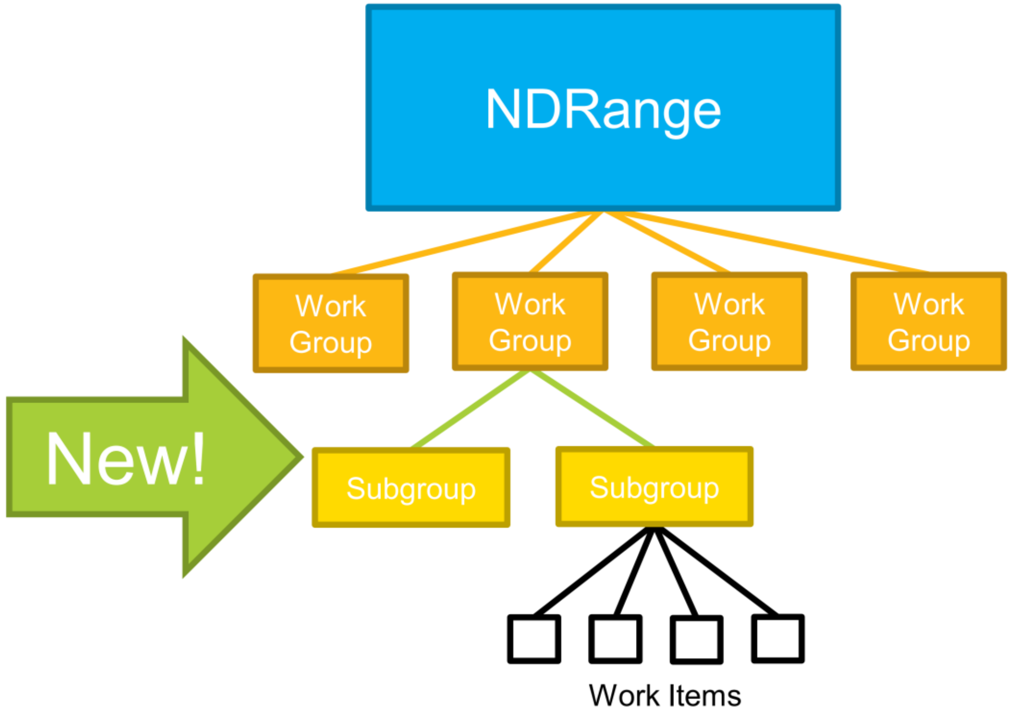
What is a Subgroup?

A Subgroup is a Collection of Work Items

- Another Level in the Execution Hierarchy
- Between Work Groups and Work Items

Key Takeaways:

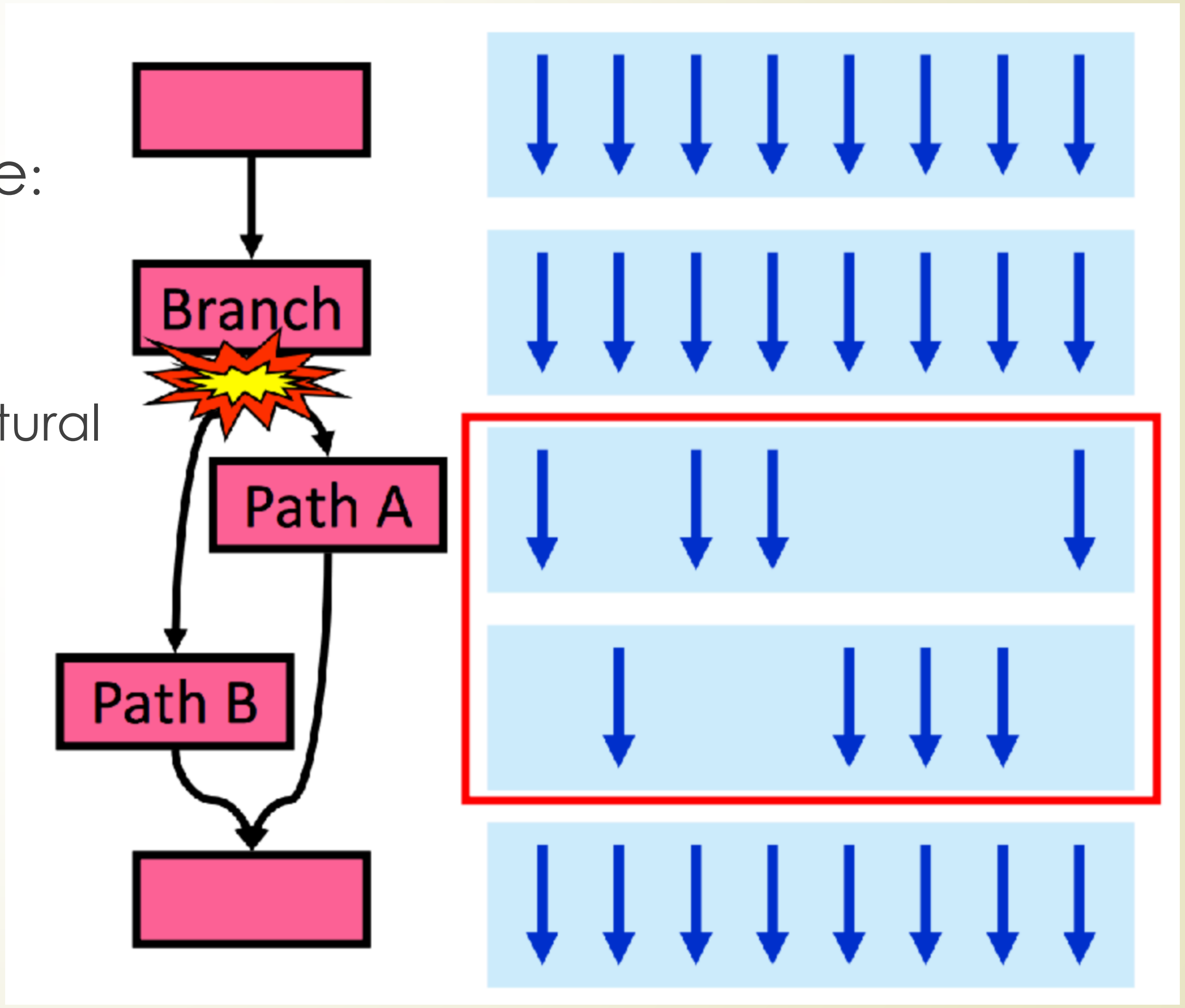
- On Intel® Processor Graphics, work items in a subgroup execute on the same EU Thread!
- Subgroups can used specialized SIMD instructions for “block operations”



Subgroup Functions bring “Explicit SIMD” to OpenCL kernels!

SIMD/SIMT Execution Degradation

- Work-group divergence:
  - Inefficient utilization
- Analogous to pipeline structural hazards and stalls
- **Strive to avoid this!**
  - **Avoid branching in kernels**



SIMD/SPMD/SIMT Summary

- Don’t worry if you feel unsure about exactly what and how these work!
- We will continue to dig into the concepts during the course.
  - Intel OpenCL extensions: subgroups

- Still lots of unknowns due to lack of documentation and vendor IP
- Opportunities for Final Project topics:
  - Microbenchmarking
  - Intel NEO Open-source driver and compiler
    - auto-scalarization and vectorization
    - Intermediate representations (IR) and native ISA

Block Matrices and Algorithms

Block Matrix Terminology

- In general we can partition both the rows and columns of an *m*-by-*n* matrix *A* to obtain

$$A = \begin{array}{c@{}c} \left[\begin{array}{ccc} A_{11} & \dots & A_{1r} \\ \vdots & & \vdots \\ A_{q1} & \cdots & A_{qr} \end{array}\right] & \begin{array}{c} m_1 \\ \\ m_q \end{array} \\ \begin{array}{ccc} n_1 & \quad\quad & n_r \end{array} & \end{array}$$

where $m_1 + \cdots + m_q = m, \qquad n_1 + \cdots + n_r = n$

and $A_{\alpha\beta}$ designates the (α, β) block (submatrix, partition, tile).

With this notation block $A_{\alpha\beta}$ has dimension $m_\alpha\mathrm{-by-}n_\beta$
and we say that $A = (A_{\alpha\beta})$ is a *q*-by-*r* block matrix.

Block Matrices and Algorithms

- Block matrix addition

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \\ A_{31} & A_{32} \end{bmatrix} + \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \\ B_{31} & B_{32} \end{bmatrix} = \begin{bmatrix} A_{11}+B_{11} & A_{12}+B_{12} \\ A_{21}+B_{21} & A_{22}+B_{22} \\ A_{31}+B_{31} & A_{32}+B_{32} \end{bmatrix}$$

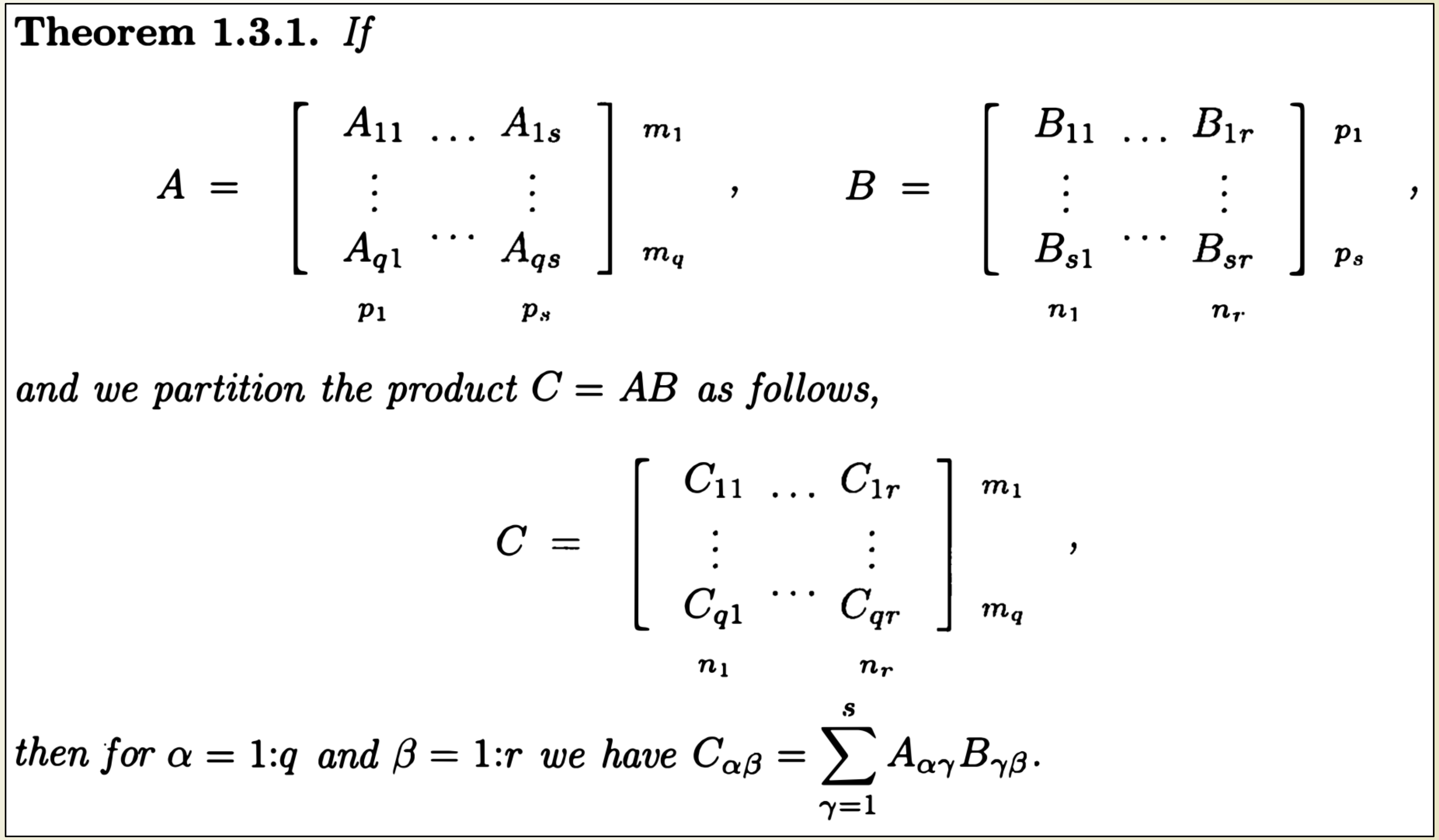
- Block matrix multiplication

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \\ A_{31} & A_{32} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} A_{11}B_{11}+A_{12}B_{21} & A_{11}B_{12}+A_{12}B_{22} \\ A_{21}B_{11}+A_{22}B_{21} & A_{21}B_{12}+A_{22}B_{22} \\ A_{31}B_{11}+A_{32}B_{21} & A_{31}B_{12}+A_{32}B_{22} \end{bmatrix}$$

  - Requires that row and column dimensions of the blocks satisfy the necessary constraints
    - Operands are ***partitioned conformably***.

Partitioned (blocked, tiled) Matrix Multiplication

- Golub & Van Loan, *Matrix Computations*, 4th, 2013



Theorem 1.3.1. *If*

$$A = \begin{array}{cc} \left[\begin{array}{ccc} A_{11} & \dots & A_{1s} \\ \vdots & & \vdots \\ A_{q1} & \cdots & A_{qs} \end{array}\right] & \begin{array}{c} m_1 \\ \\ m_q \end{array} \\ \begin{array}{ccc} p_1 & \quad\quad & p_s \end{array} & \end{array}, \qquad B = \begin{array}{cc} \left[\begin{array}{ccc} B_{11} & \dots & B_{1r} \\ \vdots & & \vdots \\ B_{s1} & \cdots & B_{sr} \end{array}\right] & \begin{array}{c} p_1 \\ \\ p_s \end{array} \\ \begin{array}{ccc} n_1 & \quad\quad & n_r \end{array} & \end{array},$$

and we partition the product $C = AB$ as follows,

$$C = \begin{array}{cc} \left[\begin{array}{ccc} C_{11} & \dots & C_{1r} \\ \vdots & & \vdots \\ C_{q1} & \cdots & C_{qr} \end{array}\right] & \begin{array}{c} m_1 \\ \\ m_q \end{array} \\ \begin{array}{ccc} n_1 & \quad\quad & n_r \end{array} & \end{array},$$

then for $\alpha = 1{:}q$ and $\beta = 1{:}r$ we have $C_{\alpha\beta} = \sum_{\gamma=1}^{s} A_{\alpha\gamma}B_{\gamma\beta}$.

Block Matrix Multiplication: Design

- So we know theoretically/analytically, we can decompose large matrices into a large number of smaller submatrices which can be computed independently.
- This is important, but ….
  - How big should the submatrix blocks be?
  - How will the data communication and memory hierarchy be used?
    - Remember
      - Each EU HW thread has limited **private** memory general-purpose register (GRF) space: 4KB
        - the amount available to each work-item depends on SW-SIMD width (subgroup size)
      - All work-items in a work-group are confined to a subslice, share 64KB **local** memory
      - The L3 cache holds data from **global** memory within the slice: 512KB for application data.
  - How will we choose the best
    - SW-SIMD width
    - NDRange dimensions, global work size, local work size?
  - How best to layout the data, based on above??

Block Matrix Mult: Intuition

OPT5 Block Matrix Mult: Algorithm

- Require all matrices (A,B,C) to be NxN (to keep it simple)
- Partition matrices into PxP blocks
- Each work-group (WG) calculates an output block
- Each work-item (WI) calculates an output block row

Kernel Pseudocode:

```
o determine current output block: WG ID [0]=row, [1]=column
o FOR P block-products of $A_{(row,0…P-1)}$ x $B_{(0…P-1,col)}$
    o Each WI put a block row $A_{(row,i)}$(WI#,:) into private memory (GRF) array
    o All WIs put block $B_{(i,col)}$(WI#,:) into local memory (SLM)
    o ---WG barrier---
    o each WI compute block product for ith row: $a_{ik}b_{kj}$
    o accumulate partial results for output row Ci in GRF array
    o ---WG barrier---
o Write final results back to global memory
```

```
// Version 0: (V0) - Initial Implementation
__attribute__((reqd_work_group_size(4,4,1)))
__kernel void mmul_opt5_tiled_v0(const int N, const int P, __global float* A, __global float* B, __global float* C)
{
        int i,k;
        int iloc = get_local_linear_id();
        int wgRow = get_group_id(0);// using dim-0 as row dim
        int wgCol = get_group_id(1);// using dim-1 as column dim
        //int glid = get_global_linear_id();

        int blkSize = N/P;
        int blkRowOffset = blkSize*wgRow;
        int blkColOffset = blkSize*wgCol;

        //printf("<mmul_opt5_tiled> glid = %d: iloc: %d, wgRow: %d, wgCol: %d, blocksize: %d, blkoffset: (row: %d, col:
        %d)\n",glid,iloc,wgRow,wgCol,blkSize,blkRowOffset,blkColOffset);

        float outtmp[16] = {0.0f};// this value needs to be updated based on N and P
        float Apriv[16] = {0.0f};// this value needs to be updated based on N and P
        local float Bloc[256];

        // each WI computes a row i of partition sub-matrix C(wgRow,wgCol)
        for(int nP=0; nP<P; nP++) // for each partition in INPUT matrices row,col
        {
                // copy global row of partition submatrix A to WI private registers
                for(k=0; k<blkSize; k++)
                        Apriv[k] = A[(blkRowOffset+iloc)*N + nP*blkSize + k];

                // copy global col of partition submatrix B to WG shared local memory (SLM)
                // WIs copy all columns of partition submatrix in parallel
                for(k=0; k<blkSize; k++)
                        Bloc[k*blkSize+iloc]= B[(nP*blkSize+k)*N + blkColOffset + iloc];

                work_group_barrier(CLK_LOCAL_MEM_FENCE); // WG sync

                // each WI compute partition row sub-product
                for(i=0; i < blkSize; i++) {
                        for(k=0; k < blkSize; k++) {
                        outtmp[i] += Apriv[k] * Bloc[k*blkSize+i];
                        }
                }
                work_group_barrier(CLK_LOCAL_MEM_FENCE); // WG sync
        }
        // write final results back to global
        for(i=0; i< blkSize; i++)
                C[(blkRowOffset+iloc)*N + blkColOffset + i] = outtmp[i];
}
```

OPT5 Block Matrix Mult: Next Steps

- We have a basic algorithm and implementation that works.
- But we're not done yet!

- Next steps:
  - profile it (get a performance baseline)
  - analyze its memory and compute performance
    - VTune!
  - Additional Optimizations

Exercise 5b: VTune Analysis of MMUL

Refer to course website: Files/InclassExercises/EX5

Lecture 5 Summary

- We looked at
  - Using OpenCL C printf( )
  - OpenCL data alignment between Host and Device
    - particularly with UDTs
    - Attribute qualifiers
  - SIMD/SPMD/SIMT and Intel HW and SW (OpenCL)
  - OPT5: Blocked Matrix Multiplication kernel (several variants)
  - Intel VTune Amplifier 2018
    - Setting up and running GPU/OpenCL Hotspot analysis
    - beginning to interpret the results for kernel performance tuning.